Disjoint Sets

Andreas Klappenecker
[using notes by R. Sekar]

Equivalence Classes

Frequently, we declare an equivalence relation on a set S. So the elements of the set S are partitioned into equivalence classes. Two equivalence classes are either the same or disjoint.

Example: S={1,2,3,4,5,6}

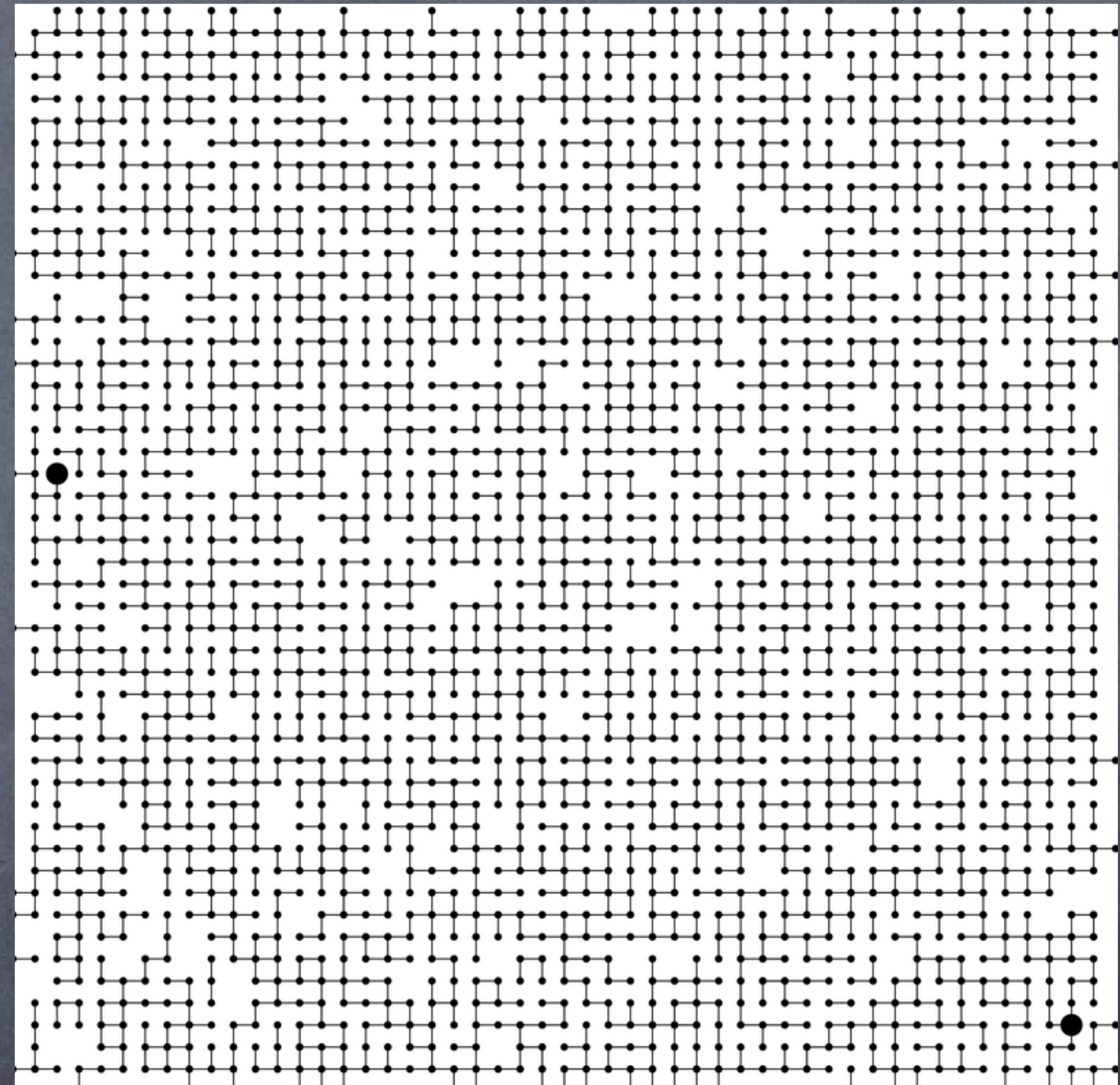
Equivalence classes: {1}, {2,4}, {3,5,6}.

The set S is partitioned into equivalence classes.

Examples

Is there a path from A to B?

Connected components form a partition of the set of nodes.



Kruskal's Minimum Spanning Tree Algorithm

The vertices are partitioned into a forest of trees.

Need: Efficient way to dynamically change the equivalence relation.

Disjoint Sets

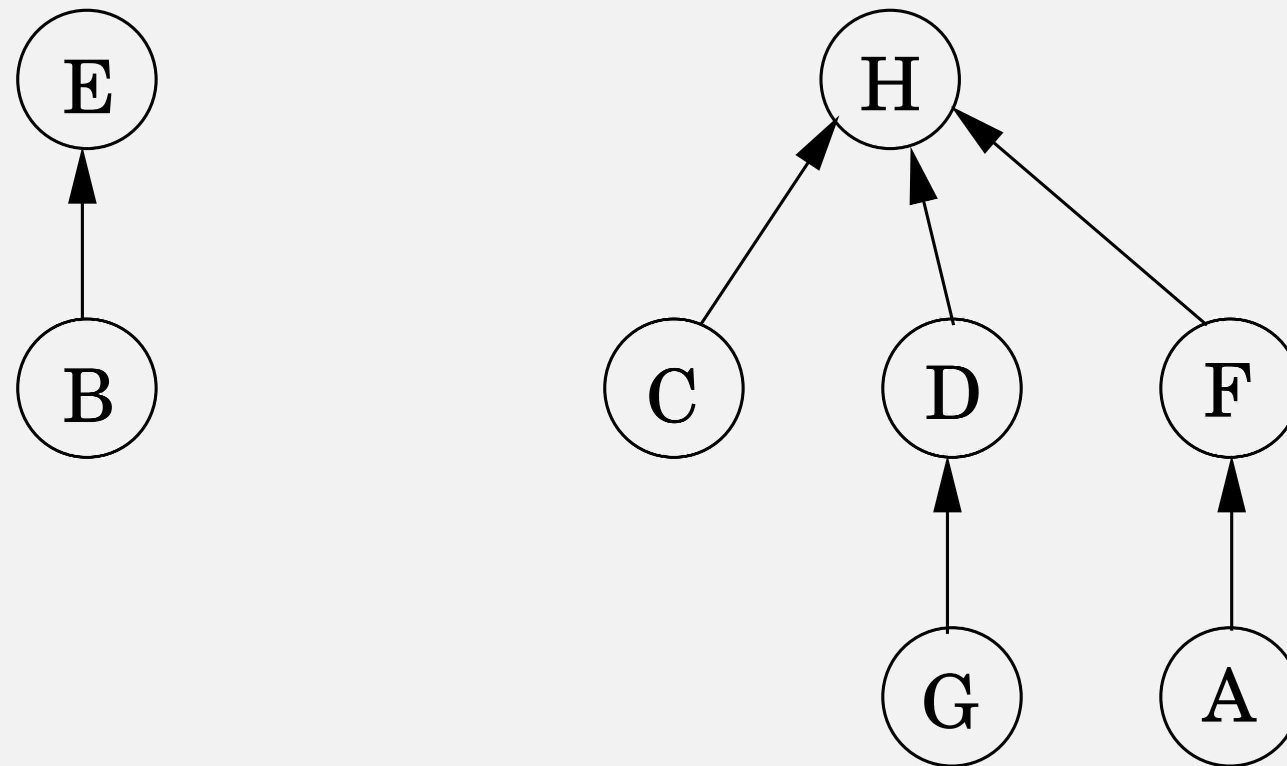
How can we represent the elements of disjoint sets?

Declare a representative element for each set.

Implementation: Inverted trees. Each element points to parent.
Root of the tree is the representative element.

Example

A directed-tree representation of two sets $\{B, E\}$ and $\{A, C, D, F, G, H\}$.



Disjoint Set Operations

Makeset(x), a procedure to form the set {x}

Find(x), a procedure to find the representative element of the set containing x.

Union(A,B), form the union of the sets A and B.

Disjoint Set Operations (Simple)

```
makeset(x)
    π(x) = x

find(x)
    while( x != π(x) ) do
        x = π(x) // find rep.
    end
return x

union(x, y)
    a = find(x)
    b = find(y)
    π(b) = a
```

Complexity of Simple Scheme

makeset(x): O(1) time

find(x): O(n) for sets of cardinality n in the worst case.

union(x): O(1) for root element, O(n) worst case.

Disjoint Set Operations (Better)

```
makeset(x)
    π(x) = x
    rank(x) = 0

find(x)
    while( x != π(x) ) do
        x = π(x)
    end
return x
```

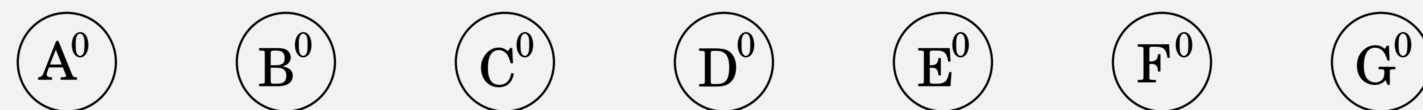
```
union(x, y)
    a = find(x); b = find(y)
    return if a = b
    if rank(a)>rank(b): π(b) = a
    else // rank(a) <= rank(b)
        π(a) = b // make b the root
        if rank(a)=rank(b): rank(b)++
```

The rank of a node is the height of the subtree rooted at that node.

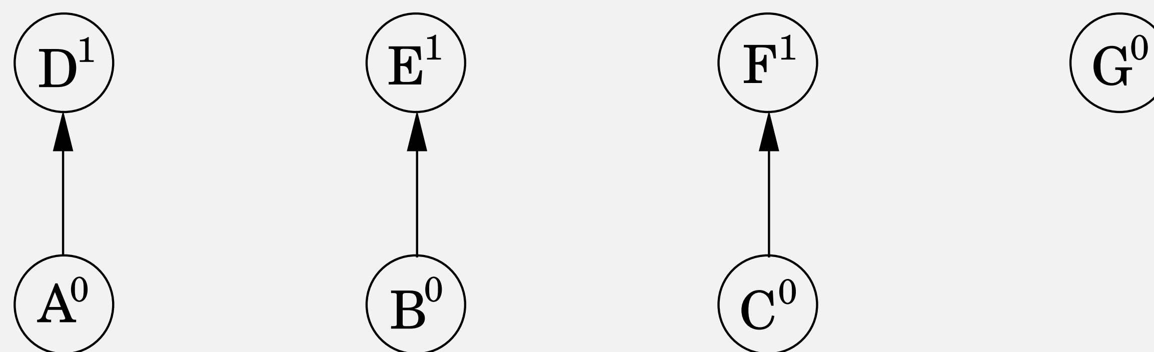
Disjoint Sets (Better)

Figure 5.6 A sequence of disjoint-set operations. Superscripts denote rank.

After makeset(A), makeset(B), $\ldots$, makeset(G):

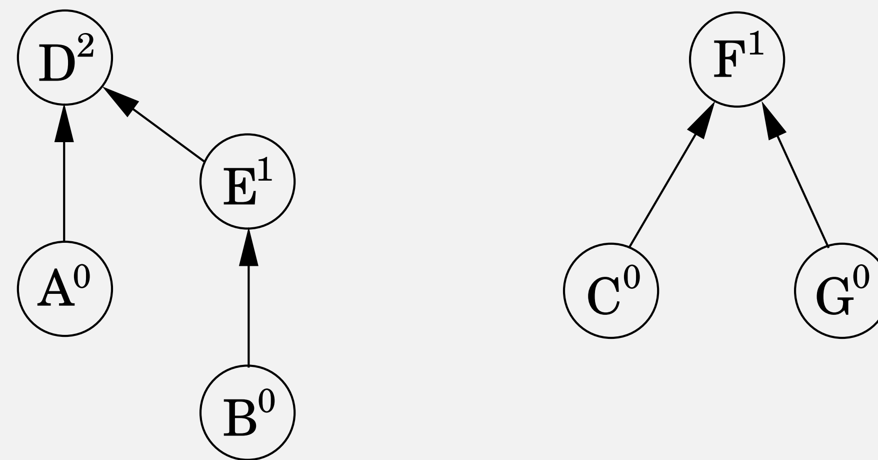


After union(A, D), union(B, E), union(C, F):

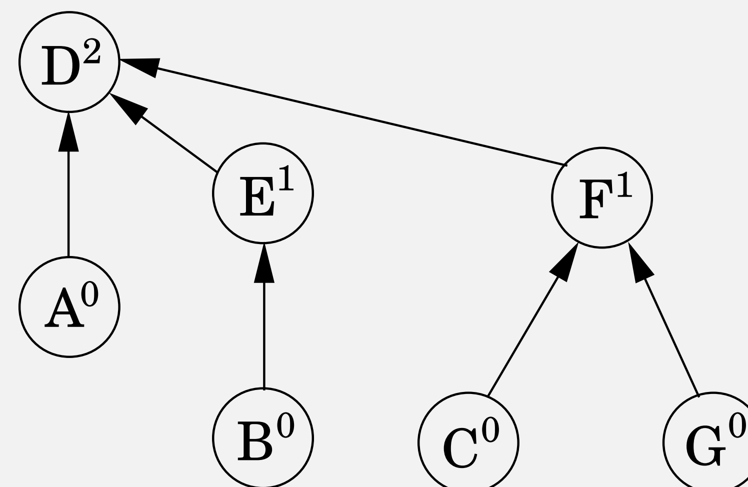


Disjoint Sets (Better)

After $\texttt{union}(C, G), \texttt{union}(E, A)$:



After $\texttt{union}(B, G)$:



Complexity of Better Scheme

makeset(x): constant time

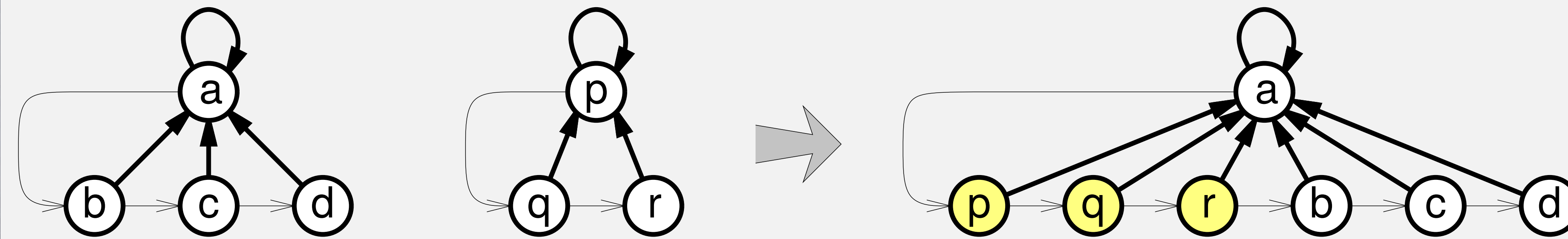
union(x,y): constant time if x and y are roots

find(x): Number of nodes of rank k never exceeds $n/2^k$. So find needs at most O(log N) time.

Improving Find

Idea: Why not force depth to be 1? Then `find` will have $O(1)$ complexity!

Approach: **Threaded Trees**



Problem: Worst-case complexity of `union` becomes $O(n)$

Solution:

- Merge smaller set with larger set
- Amortize cost of `union` over other operations

Disjoint Sets w/ Threaded Trees

- Other than cost of updating parent pointers, `union` costs $O(1)$
- **Idea:** Charge the cost of updating a parent pointer to an element.
- **Key observation:** Each time an element's parent pointer changes, it is in a set that is twice as large as before
  - So, with n operations, you can at most $O(\log n)$ parent pointer updates per element
- Thus, amortized cost of n operations, consisting of some mix of `makeset`, `find` and `union` is at most $n \log n$

Quo Vadis?

Threaded trees are better for find, but not so great for union.

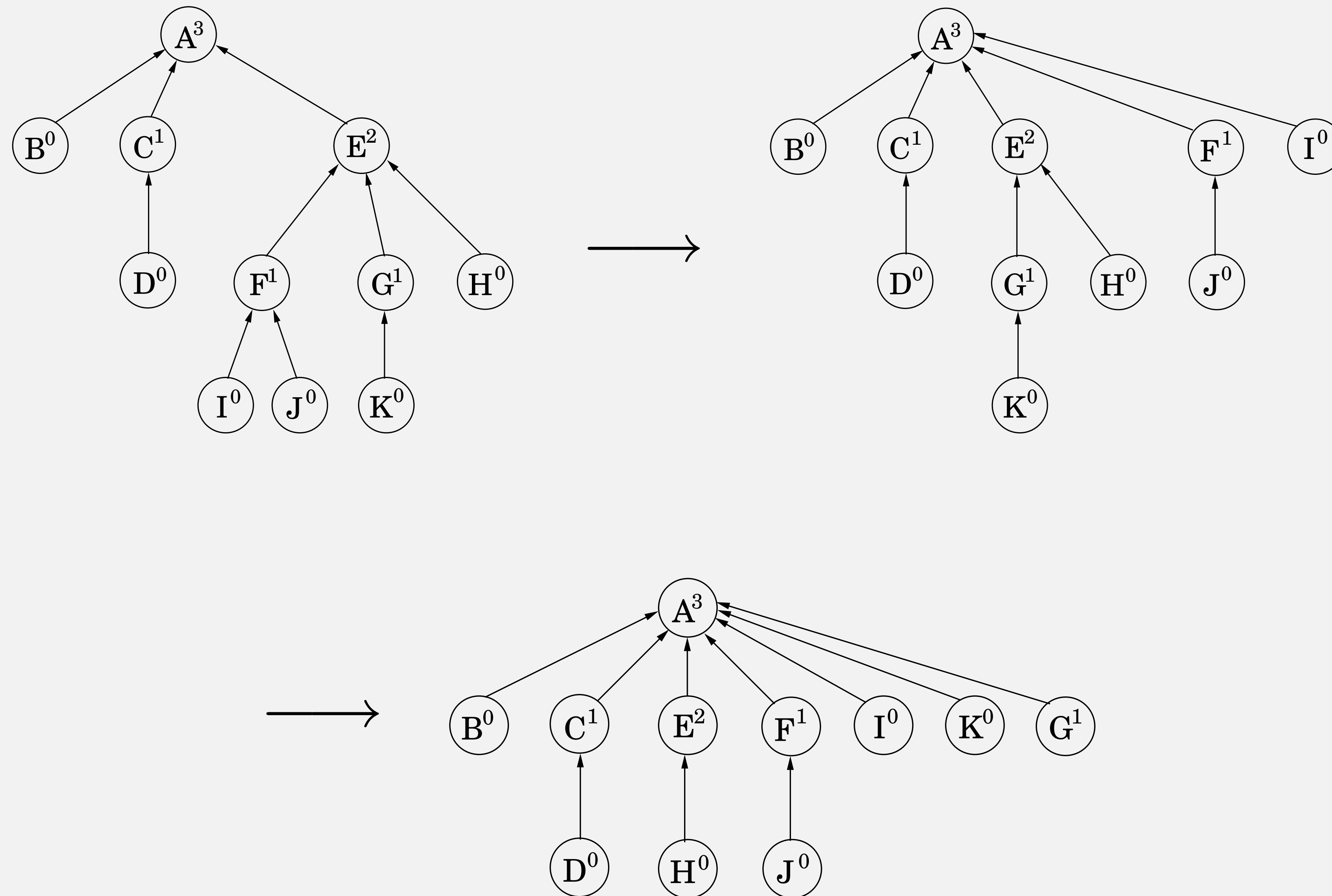
The previous scheme was better for union, but not so great for find.

Can we formulate an eager approach for find and a lazy approach for union, getting the best of both worlds?

Path Compression

- **Path compression:** Retains *lazy* `union`, but when a `find(x)` is called, *eagerly* promotes x to the level beloe the root
  - Actually, we promote $x, \pi(x), \pi(\pi(x)), \pi(\pi(\pi(x)))$ and so on.
  - As a result, subsequent calls to find x or its parents become cheap.
- From here on, we let *rank* be defined by the *union* algorithm
  - For root node, *rank* is same as depth
  - But once a node becomes a non-root, its *rank* stays fixed,
    - *even when path compression decreases its depth.*

Example



Log^*

Amortized cost per operation of n set operations is $O(\log^* n)$ where

$$\log^* x = \text{smallest } k \text{ such that } \underbrace{\log(\log(\cdots \log}_{k \text{ times}}(x)\cdots)) = 1$$

Note: $\log^*(x) \leq 5$ for virtually any n of practical relevance.

Specifically,

$$\log^*(2^{65536}) = \log^*(2^{2^{2^{2^2}}}) = 5$$

Note that 2^{65536} is approximately a $20,000$ digit decimal number.

We will never be able to store input of that size, at least not in our universe. (Universe contains may be 10^{100} elementary particles.)

So, we might as well treat $\log^*(n)$ as $O(1)$.

Path Compression

- For n operations, *rank* of any node falls in the range $[0, \log n]$
- Divide this range into following groups:

$$[1], [2], [3\text{–}4], [5\text{–}16], [17\text{–}2^{16}], [2^{16}+1\text{–}2^{65536}], \ldots$$

  Each range is of the form $[k\text{–}2^{k-1}]$

- Let $G(v)$ be the group $rank(v)$ belongs to: $G(v) = \log^*(rank(v))$
- Note: when a node becomes a non-root, its rank never changes

Key Idea

Give an "allowance" to a node when it becomes a non-root. This allowance will be used to pay costs of path compression operations involving this node.
For a node whose rank is in the range $[k\text{–}2^{k-1}]$, the allowance is 2^{k-1}.

Amortized Cost

- Recall that number of nodes of rank r is at most $n/2^r$
- Recall that a node of rank in the range $[k\text{–}2^{k-1}]$ is given an allowance of 2^{k-1}.
- Total allowance handed out to nodes with ranks in the range $[k\text{–}2^{k-1}]$ is therefore given by

$$2^{k-1}\left(\frac{n}{2^k} + \frac{n}{2^{k+1}} + \cdots + \frac{n}{2^{2^{k-1}}}\right) \leq 2^{k-1}\frac{n}{2^{k-1}} = n$$

- Since total number of ranges is $\log^* n$, total allowance granted to all nodes is $n \log^* n$
- We will spread this cost across all n operations, thus contributing $O(\log^* n)$ to each operation.

Amortized Cost 2

- Cost of a `find` equals # of parent pointers followed
  - Each pointer followed is updated to point to root of current set.
- **Key idea:** Charge the cost of updating $\pi(p)$ to:
  - *Case 1:* If $G(\pi(p)) \neq G(p)$, then charge it to the current `find` operation
    - Can apply only $\log^* n$ times: a leaf's G-value is at least 1, and the root's G-value is at most $\log^* n$.
    - Adds only $\log^* n$ to cost of `find`
  - *Case 2:* Otherwise, charge it to p's allowance.
    - Need to show that we have enough allowance to to pay each time this case occurs.

Amortized Cost 3

- If $\pi(p)$ is updated, then the rank of p's parent increases.
- Let p be involved in a series of `find`'s, with q_i being its parent after the ith `find`. Note

$$rank(p) < rank(q_0) < rank(q_1) < rank(q_2) < \cdots$$

- Let m be the number of such operations before p's parent has a higher G-value than p, i.e., $G(p) = G(q_m) < G(q_{m+1})$.
- Recall that
  - A $G(p) = r$ then r corresponds to a range $[k\text{-}2^{k-1}]$ where $k \leq rank(p) \leq 2^{k-1}$. Since $G(p) = G(q_m)$, $q_m \leq 2^{k-1}$
  - The allowance given to p is also 2^{k-1}

  So, there is enough allowance for all promotions up to m.
- After $m+1$th `find`, the `find` operation will pay for pointer updates, as $G(\pi(p)) > G(p)$ from here on.